

Lab1: 简单集群搭建

1. 概况

(((截图太多了因为全是配置过程 我下个报告尽可能改🌊🌊🌊

实验环境: macOS 26.5.1, Debian GNU Linux

使用到的Agent: GitHub Copilot/Codex

模型: ChatGPT-5.x & GLM-5.2 & deepseek v4-pro

通过此次环境配置和小集群的搭建,我意识到了AI的日益强大,也意识到自己亟需掌握更多Linux环境的知识和经验。文档的信息量不小,我需要慢慢消化。

于 2026.7.22 更新

2. 任务一: 从源码构建 OpenMPI、BLAS 和 HPL

2.1 OpenMPI的构建

- 下载并 `scp` 压缩包

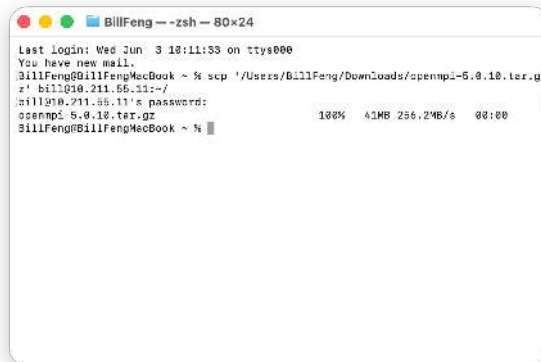


Image 1

- 解压并安装

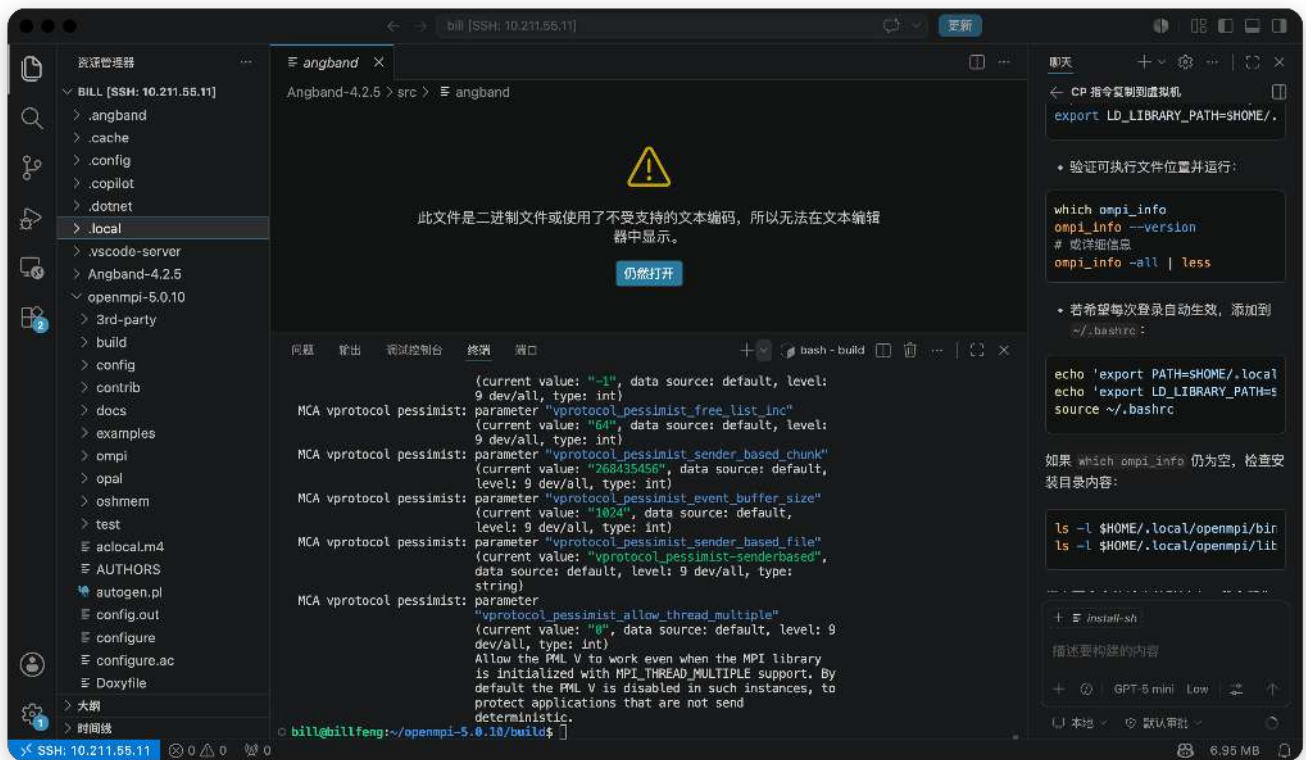


Image 2

2.2 BLAS&CBLAS

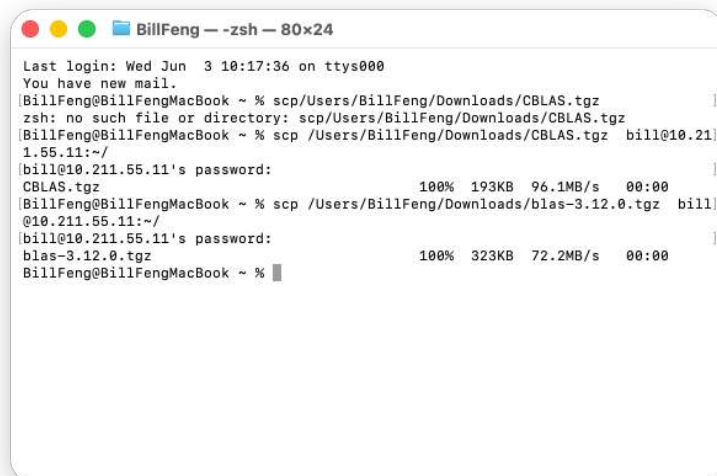


Image 3

- 修改编译器配置:

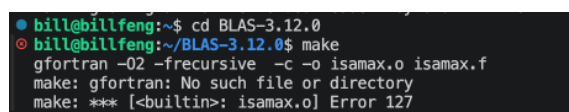


Image 4

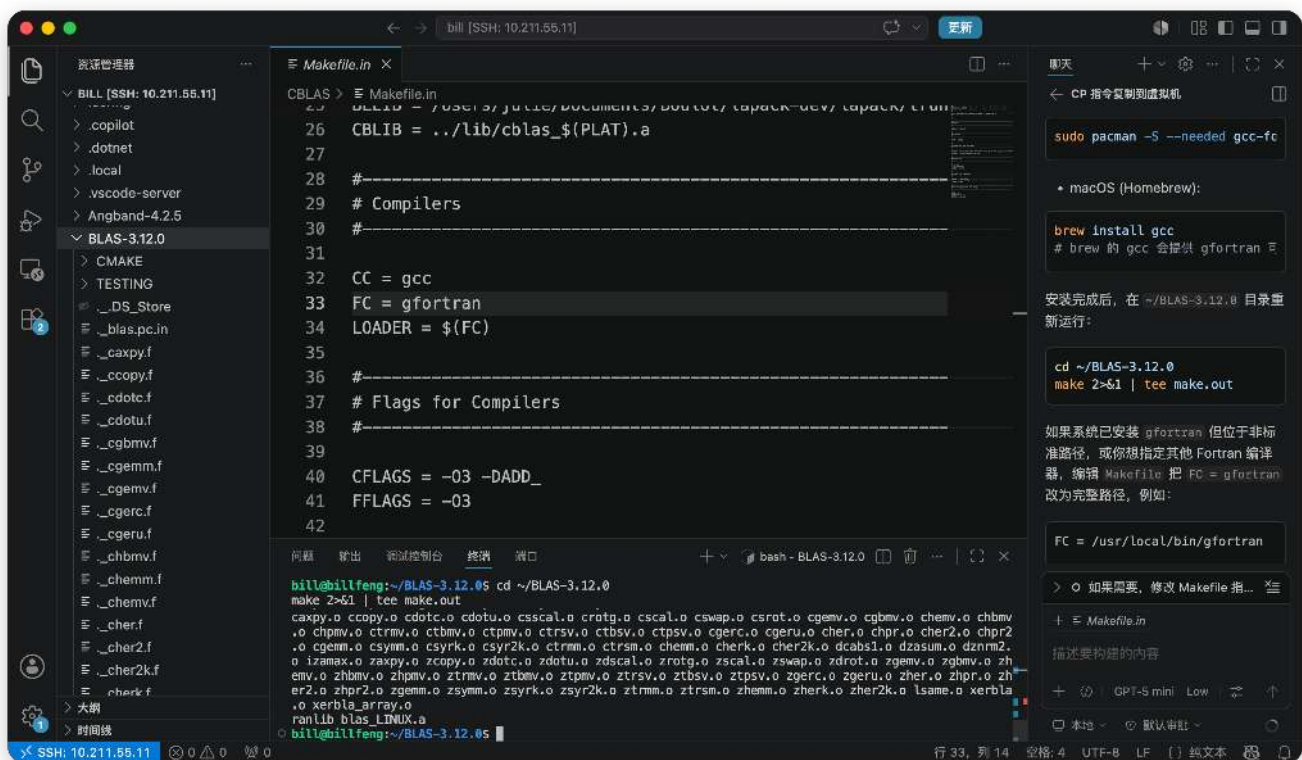


Image 5

- 遇到了文档中的问题，修改 TOPdir 路径：

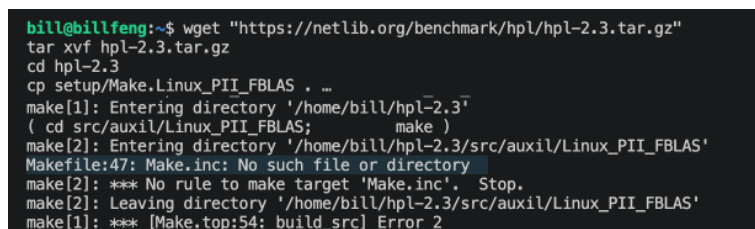
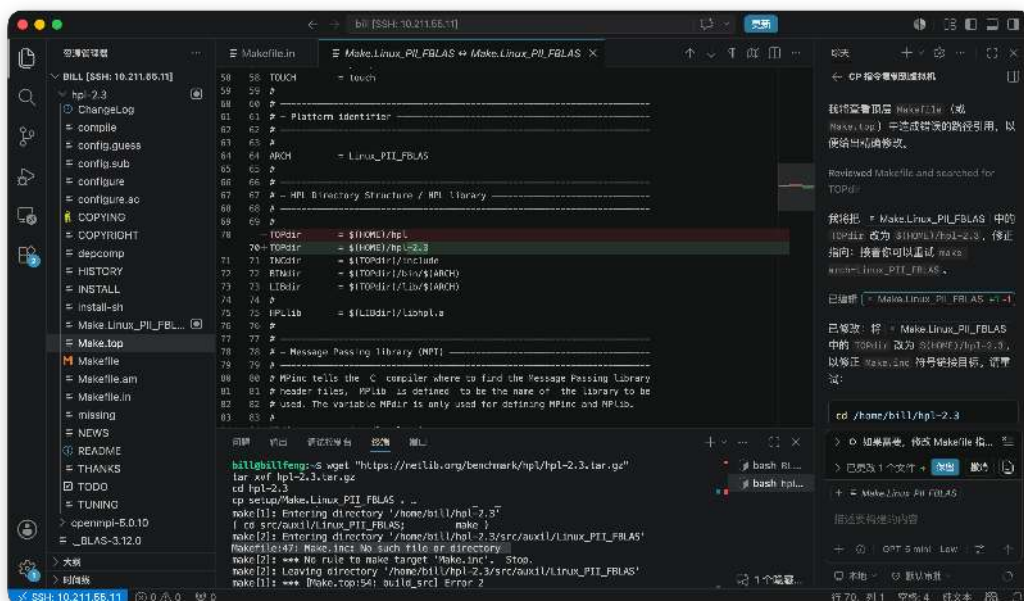


Image 6



2.3 HPL

下面是在 `Make.Linux_PII_FBLAS` 中修改或设定的参数：

- `TOPdir` : `$(HOME)/hpl-2.3`
- `MPdir` : ``/home/bill/.local/openmpi`
- `MPinc` : ``-I$(MPdir)/include`
- `MPLib` : `$(MPdir)/lib/libmpi.so`
- `LAdir` : `$(HOME)/BLAS-3.12.0`
- `LALib` : `$(LAdir)/blas_LINUX.a $(LAdir)/cblas_LINUX.a`
- `LINKER` : `/usr/bin/gfortran`

成功生成可执行文件在 `bin/Linux_PII_FBLAS` 目录下：

```
bill@billfeng:~/hpl-2.3$ ls bin/Linux_PII_FBLAS/
HPL.dat  xhpl
```

Image 8

3. 任务二：配置 NFS 共享目录

- 这是四个节点映射的IP地址：

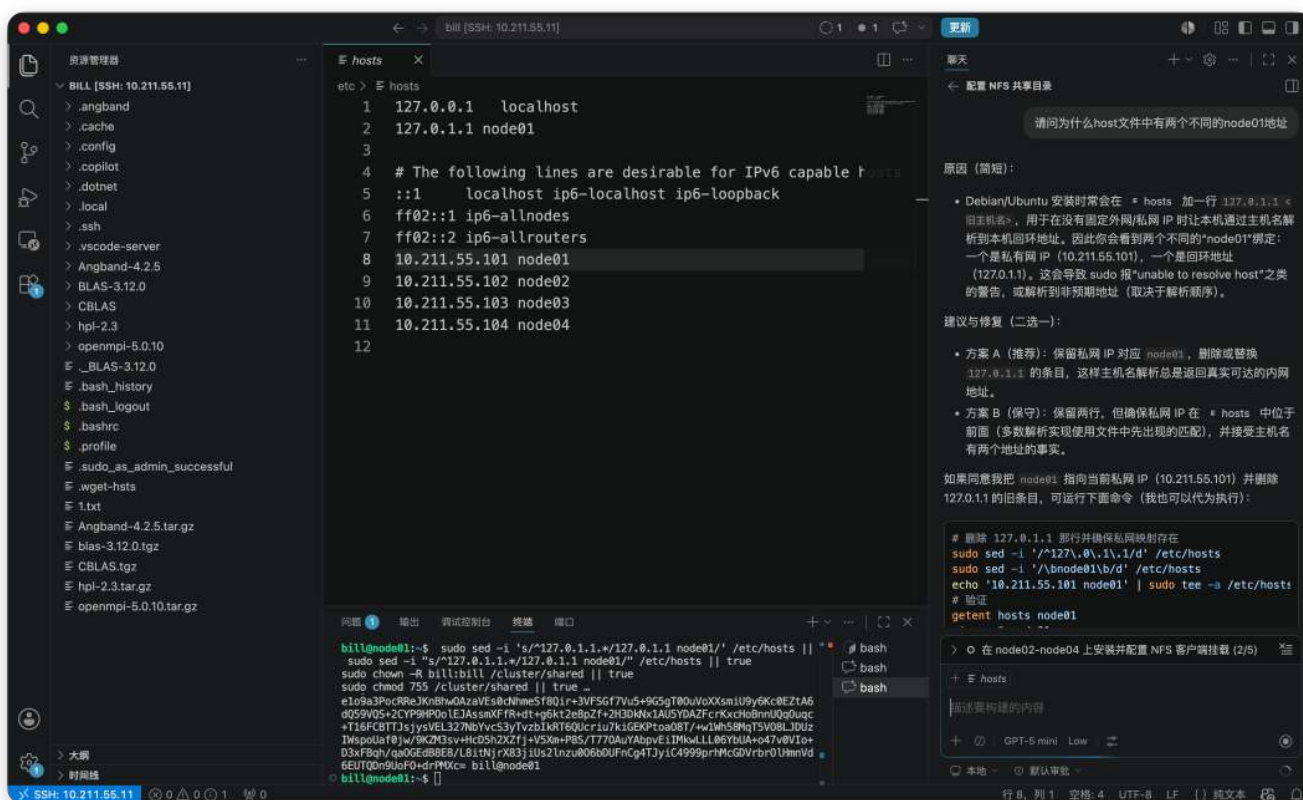


Image 9

- 我的思路是先配置 `node02`, 然后将配置好的节点拷贝两份。下面三张图是我在 `node02` 中操作的记录。

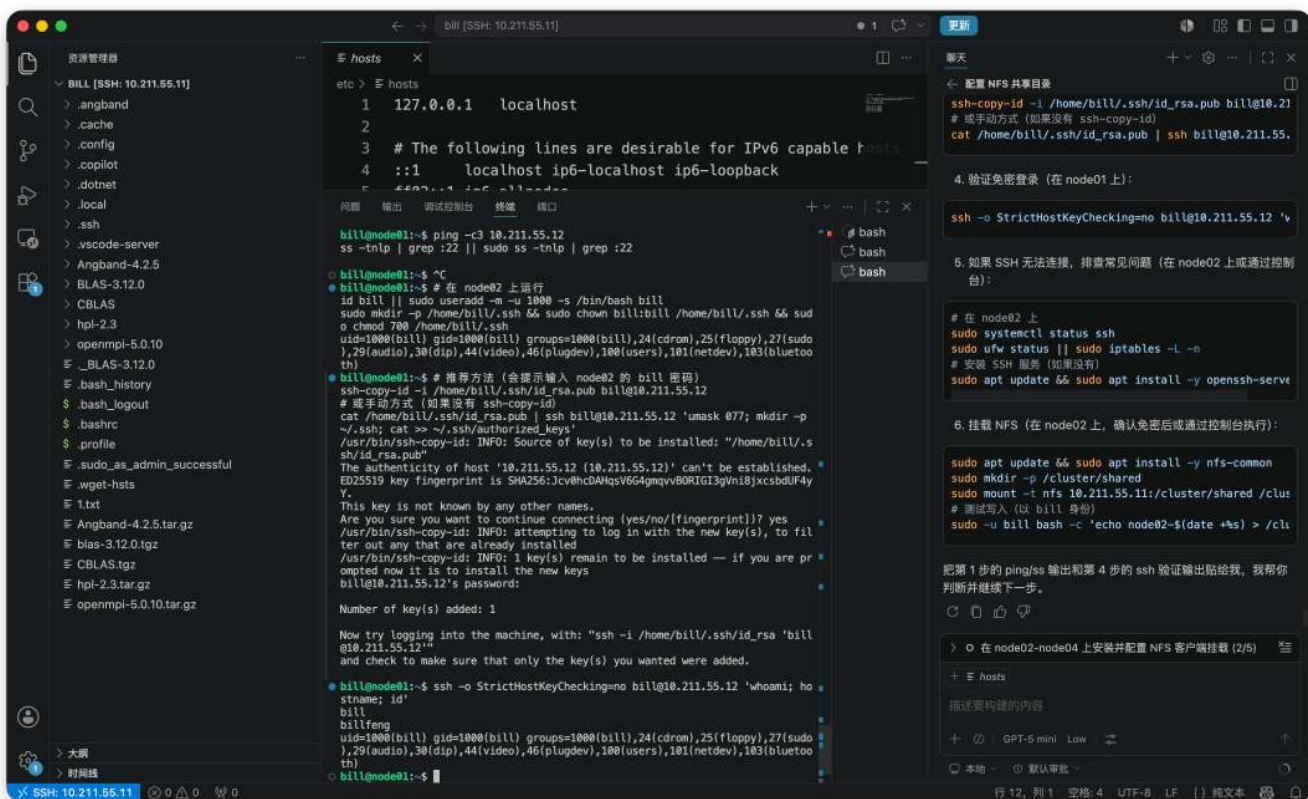


Image 10

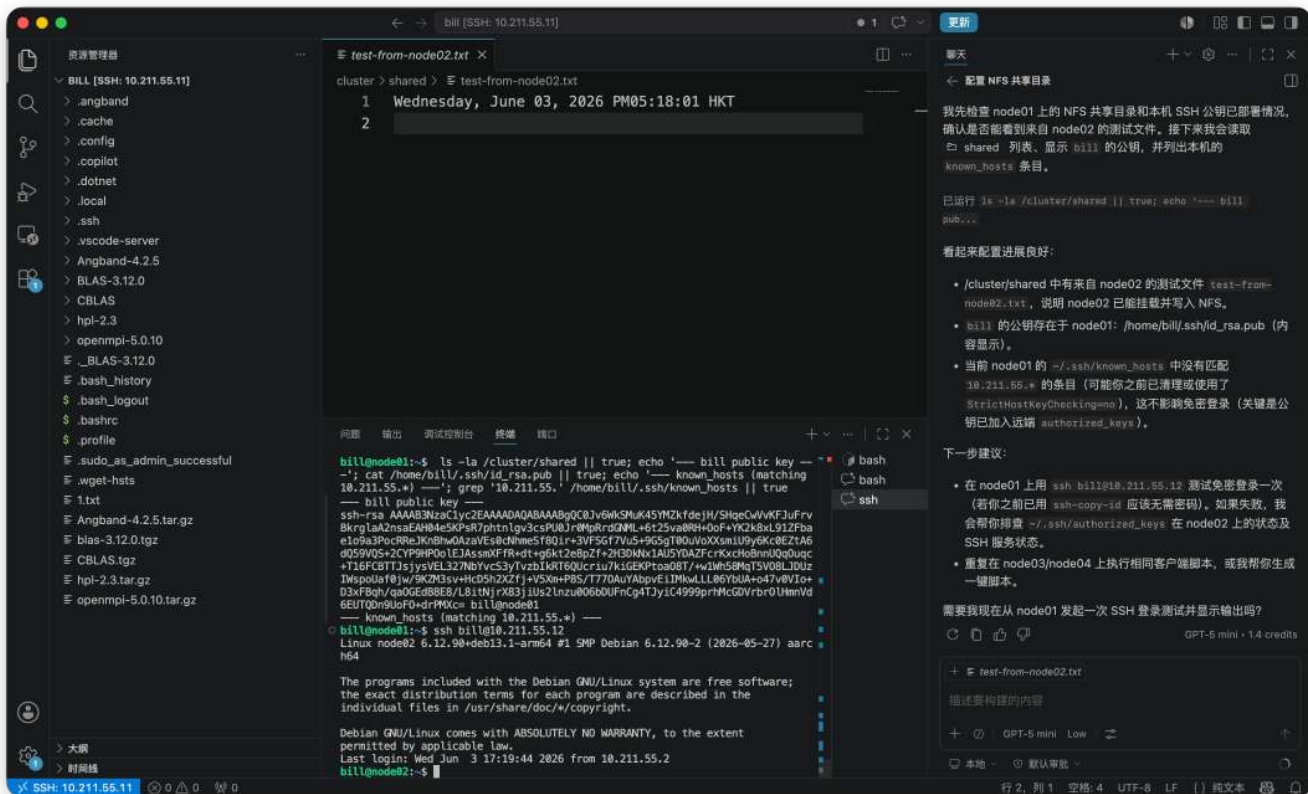


Image 11

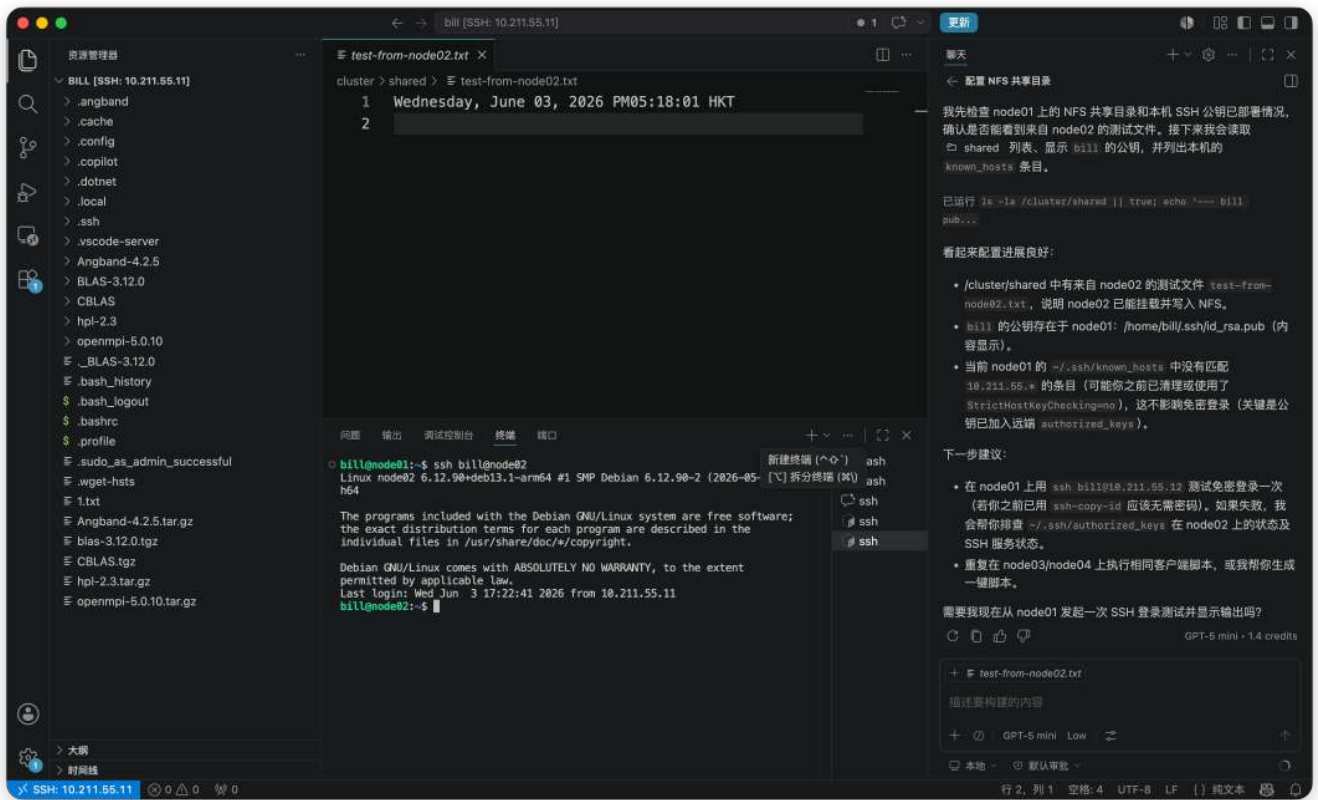


Image 12

- 将密钥生成算法修改为更推荐的 `ed25519`，并在此处清理了多余的密钥。
- 在此处我还保持着对AI的掌控，尽量不用 `Agent` 而是用 `Ask` 模式，亲自输入或者粘贴命令，了解每一步都在做什么。到任务三点时候就有点“破戒”了。

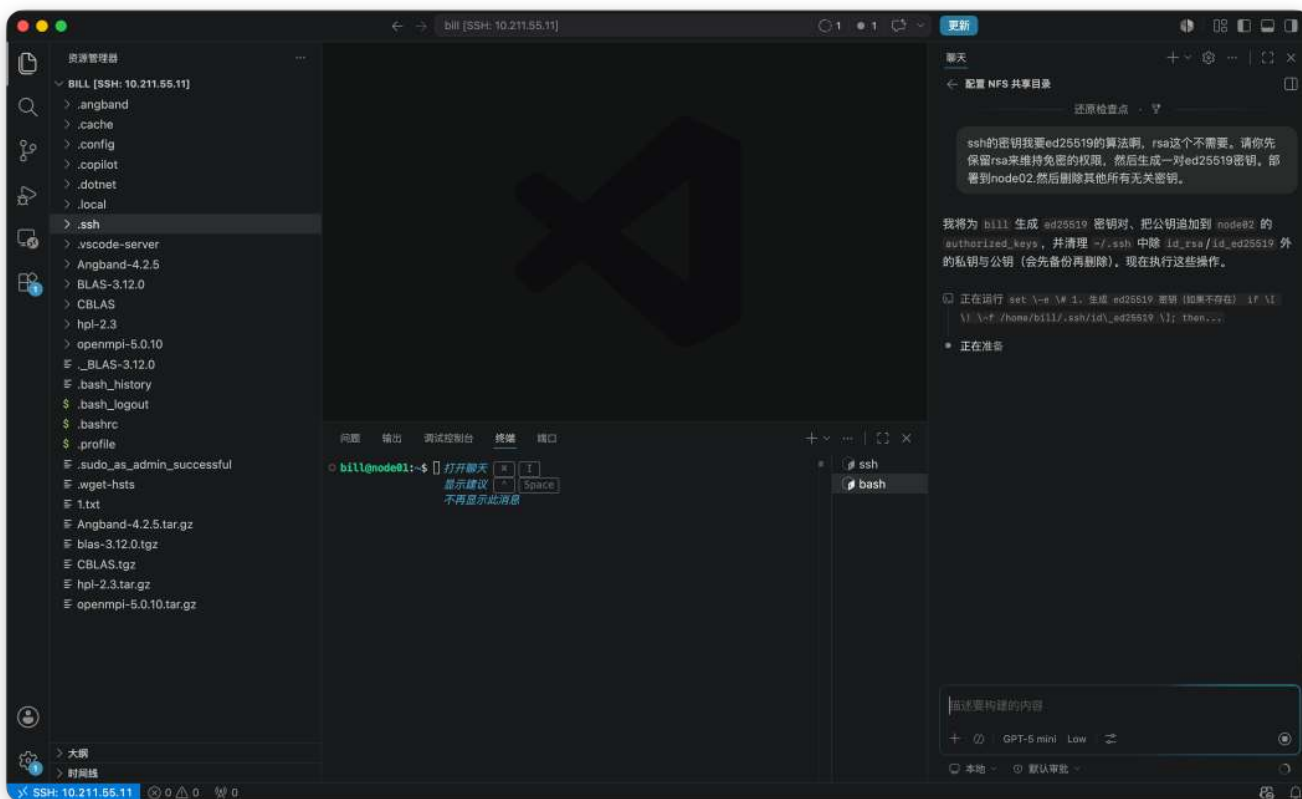


Image 13

- 见图片批注

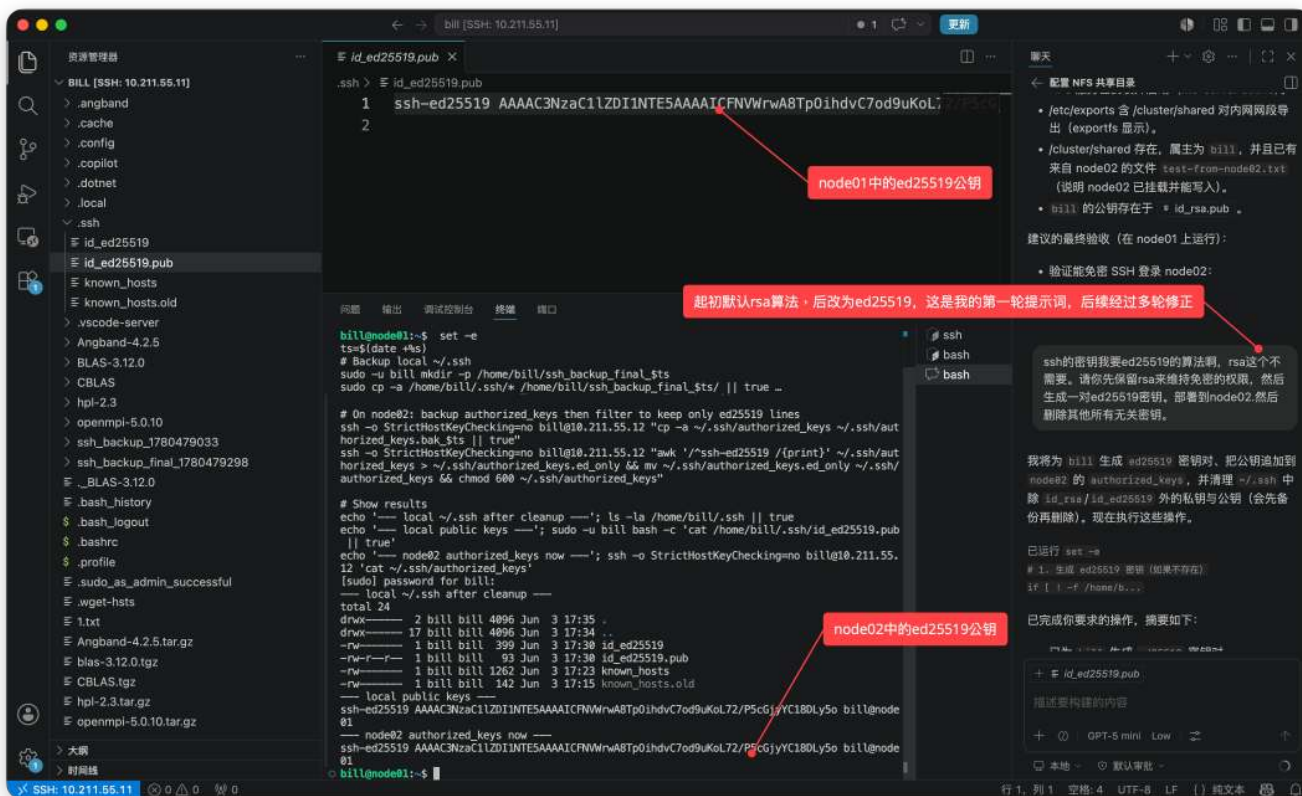


Image 14

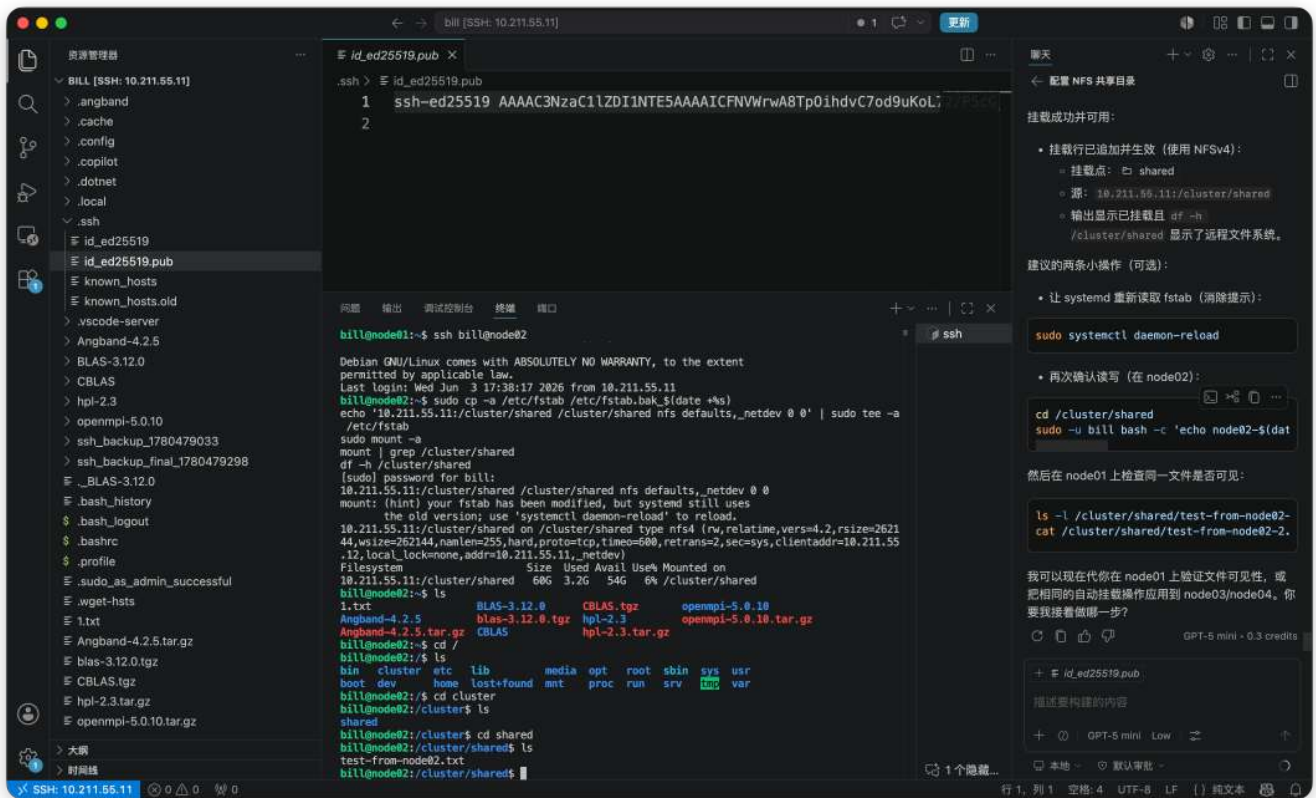


Image 15

- 拷贝 `node03 node04`，配置其地址和主机名。
- 4节点读写测试成功:

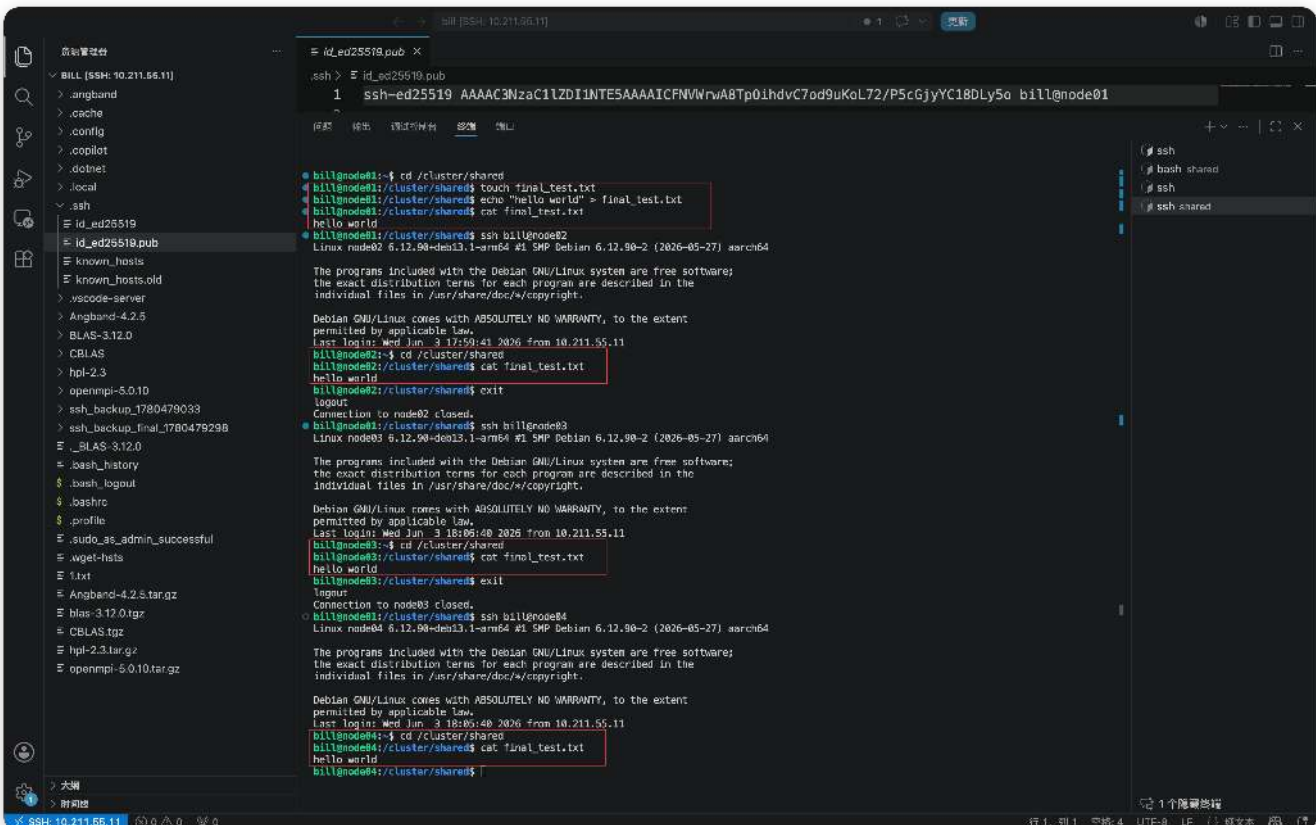


Image 16

- 读写文件时只和 `node01` 通信，`node02` 不通过 `node03` 读文件。

```

1 | node02 ┌
2 | node03 ┤ node01 (NFS 服务端)
3 | node04 └
4 | //有点累，吃饭去了，明天继续。

```

4. 任务三：配置 Slurm 作业调度系统

- 集群搭建后 `srun` 单节点正常，但多节点作业报错。
- 我开始意识到自己的能力比较局限，借用了 `Agent` 来排查问题，发现是因为 `UID 不一致` 导致 `Slurm` 跨节点通信失败

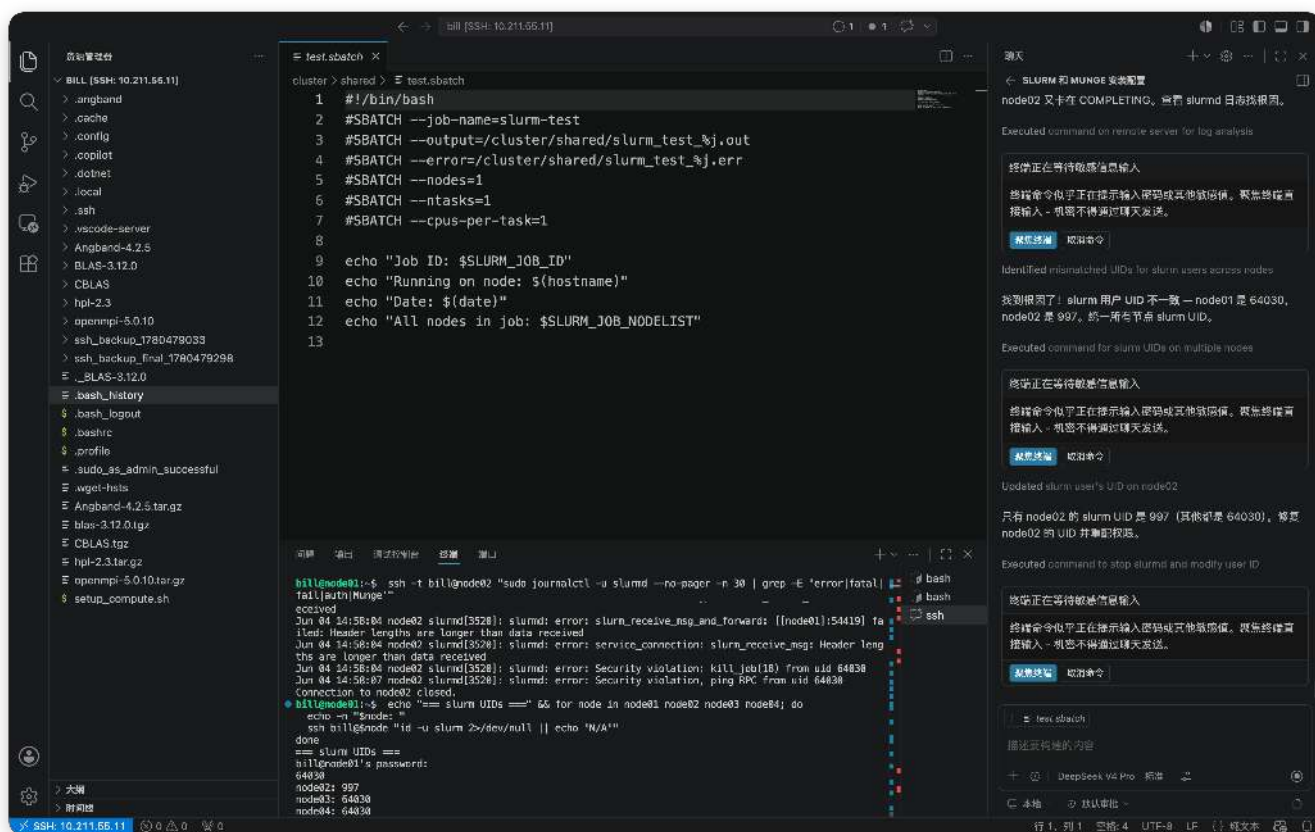


Image 17

修改 uid 一致后成功通信

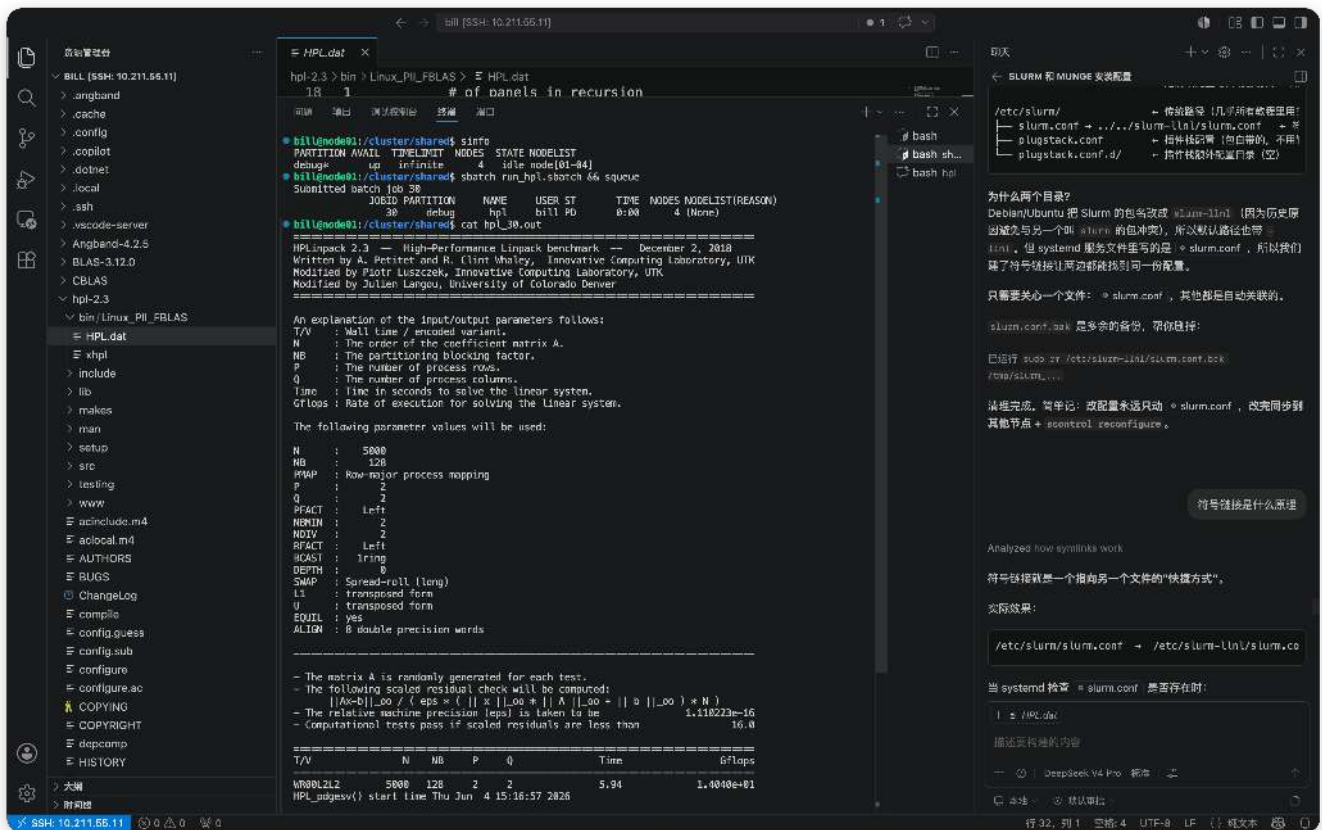


Image 18

5. 任务四：使用 Slurm 提交 HPL

- 关于 `HPL.dat` :

把我们刚才关于dat的关键对话的精华总结一下

Reviewed key points about HPL.dat

HPL.dat 核心总结

文件中的格式: 数量 + 具体的值

例:

1	= of NBs	+ 1 个 NB 值
128	NBs	= 值 = 128, 分块大小

三个关键参数:

参数	含义	你的值	规则
N	矩阵是 N×N	10000	越大计算越多, 受内存限制 (N²×8 字节)
NB	分块粒度	128	影响缓存效率, 通常 32~256
P, Q	进程网格的行列	2, 2	P × Q = 进程数, 尽量接近正方形

P×Q 与 Slurm 的关系 (最容易搞错):

Slurm: `--ntasks=4` → 4 个 MPI 进程
HPL.dat: `P×Q=2×2=4` → 必须相等!

不寄会怎样:

`P×Q < ntasks` → 浪费资源
`P×Q > ntasks` → HPL 报错

HPL 会遍历所有参数组合:

1 个 N × 1 个 NB × 1 组 P×Q = 跑 1 次
3 个 N × 2 个 NB × 2 组 P×Q = 跑 12 次

DeepSeek V4 Pro · 0x

Image 19

- 根据文档推荐，去掉 node01 的计算节点属性，此时 node01 是登陆和控制节点，node02-04 是计算节点。

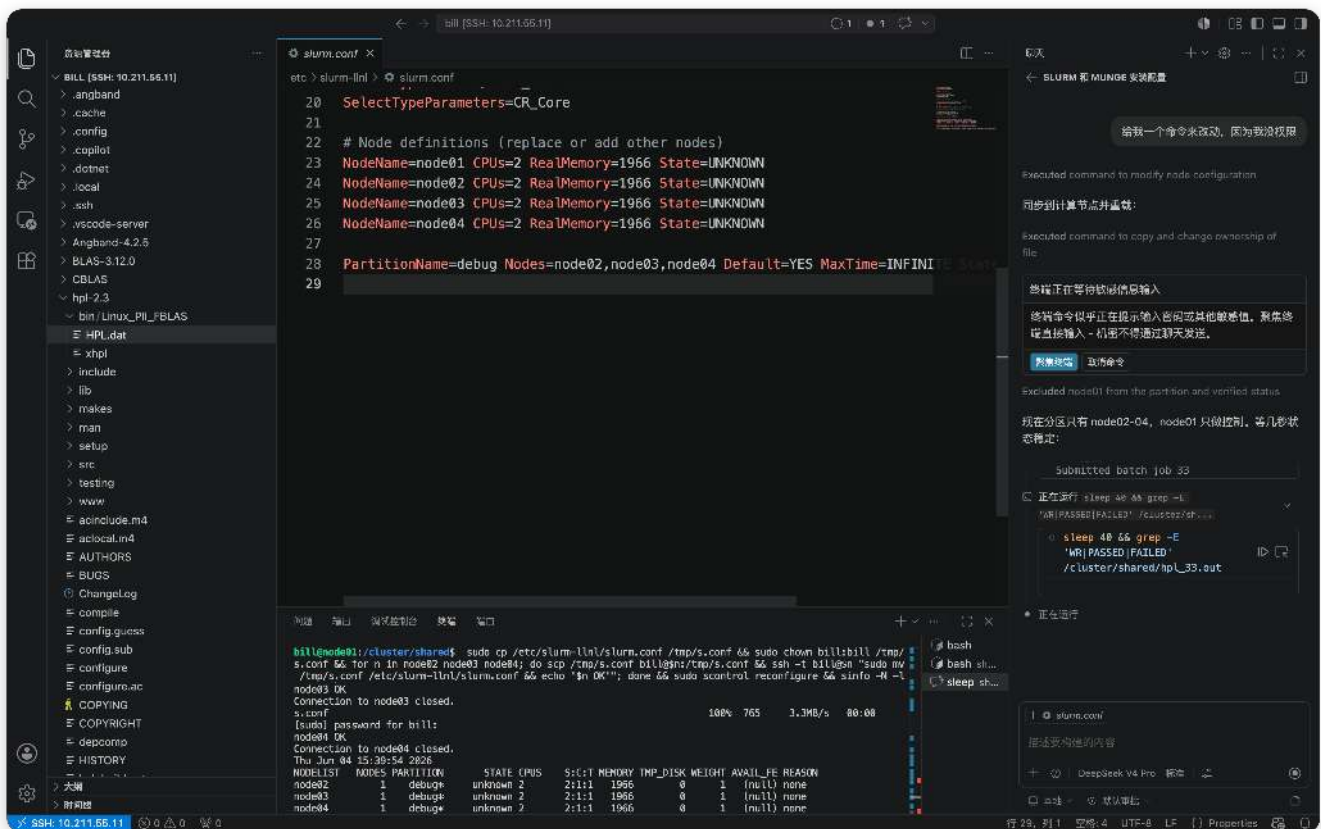


Image 20

- 用于提交HPL的脚本 `run hpl.sbatch`

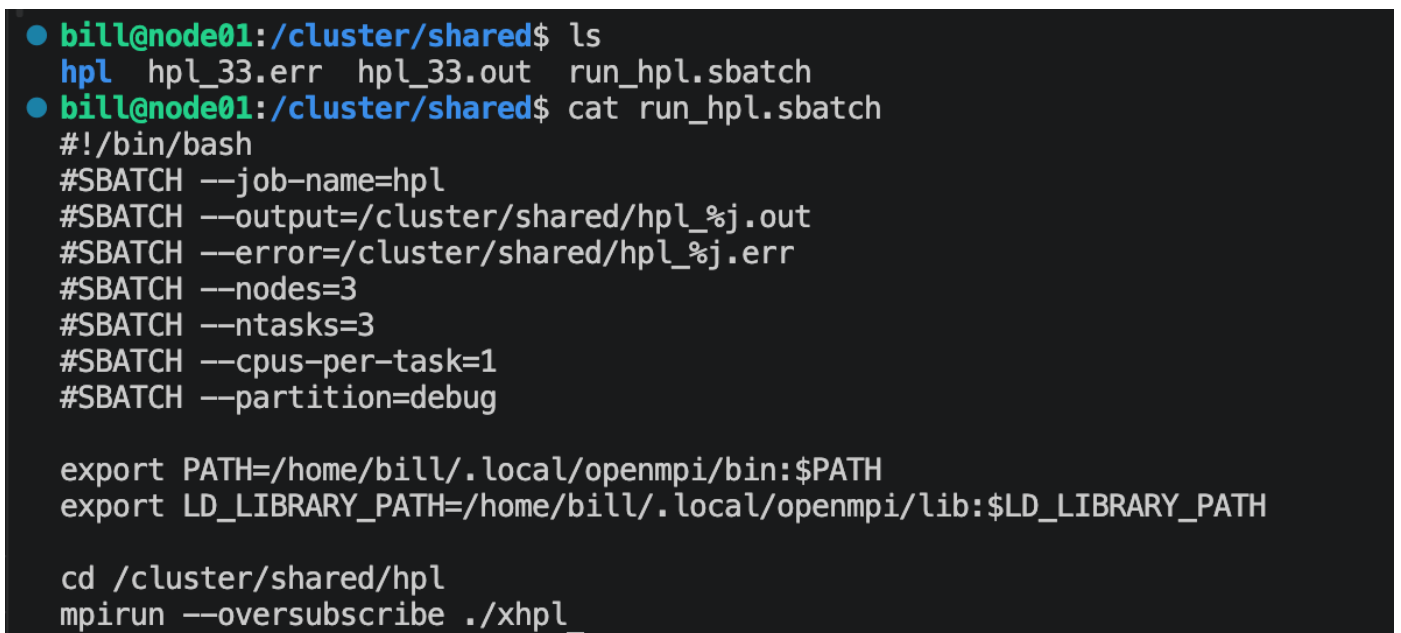


Image 21

- 见图片批注。

```

bill@node01:/cluster/shared$ cat HPL.dat
cat: HPL.dat: No such file or directory
bill@node01:/cluster/shared$ cat hpl/HPL.dat
cat: hpl/HPL.dat: No such file or directory
bill@node01:/cluster/shared$ cat hpl/HPL.dat
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
6            device out (0=stdout,7=stderr,file)
1            # of problems sizes (N)
10000       N
1            # of NBs
128         NB
0           PMAP process mapping (0=Row-,1=Column-major)
1           # of process grids (P x Q)
1           Ps
3           Qs
16.0        threshold
1           # of panel fact
0           PFACTs (0=left, 1=Crout, 2=Right)
1           # of recursive stopping criterium
2           NBMINS (>= 1)
1           # of panels in recursion
2           NDIVs

```

矩阵规模

分块粒度

node01作为控制节点，node02-04作为计算节点。三个计算节点只能分配1*3的条形进程网络

Image 22

- 随后通过任务管理器查看CPU资源情况，发现资源在 node02 上堆满，而其他节点几乎没有任务。

0.49
0.76为什么主机显示node02吃满了而其他的没有

Analyzed HPL workload distribution across nodes

因为 mpirun 没有跟 Slurm 通信，它把所有 3 个进程全丢在 node02 上了，node03/node04 在空转。修一下 sbatch 脚本：

已运行 `cat > /cluster/shared/run_hpl.sbatch << 'EOF'`

```
#!/b...
```

Updated script to specify nodes for OpenMPI usage

原因解释：

之前：mpirun --oversubscribe ./xhpl

- OpenMPI 不知道 Slurm 分了哪些节点
- 三个进程全挤在 node02，node03/node04 闲着

现在：mpirun --host \$SLURM_JOB_NODELIST ./xhpl

- Slurm 自动填成 mpirun --host node02,node03,node04 ./xhpl
- 三个进程真正分发到三台机器

提交：

sbatch run_hpl.sbatch && queue

DeepSeek V4 Pro · 0x

Image 23

- 来自AI的解释：`$SLURM_JOB_NODELIST` 返回压缩格式 `node[02-04]`，OpenMPI不认得。展开它。
- 修改后性能有所提升，见下图：
 - 上面框内部分是 node02 单独计算

- 下面框内部分是 node02-04 协同计算

```

- The relative machine precision (eps) is taken to be 1.110223e-16
- Computational tests pass if scaled residuals are less than 16.0

=====
T/V          N    NB    P    Q          Time          Gflops
=====
WR00L2L2     10000  128    1    3          50.97          1.3083e+01
HPL_pdgesv() start time Thu Jun  4 15:49:16 2026

HPL_pdgesv() end time   Thu Jun  4 15:50:07 2026

=====
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)= 4.51323610e-03 ..... PASSED
=====

- The matrix A is randomly generated for each test.
- The following scaled residual check will be computed:
  ||Ax-b||_oo / ( eps * ( || x ||_oo * || A ||_oo + || b ||_oo ) * N )
- The relative machine precision (eps) is taken to be 1.110223e-16
- Computational tests pass if scaled residuals are less than 16.0

=====
T/V          N    NB    P    Q          Time          Gflops
=====
WR00L2L2     10000  128    1    3          39.83          1.6741e+01
HPL_pdgesv() start time Thu Jun  4 15:53:00 2026

```

Image 24

- 任务管理器的资源分配说明进程调度是有效的！

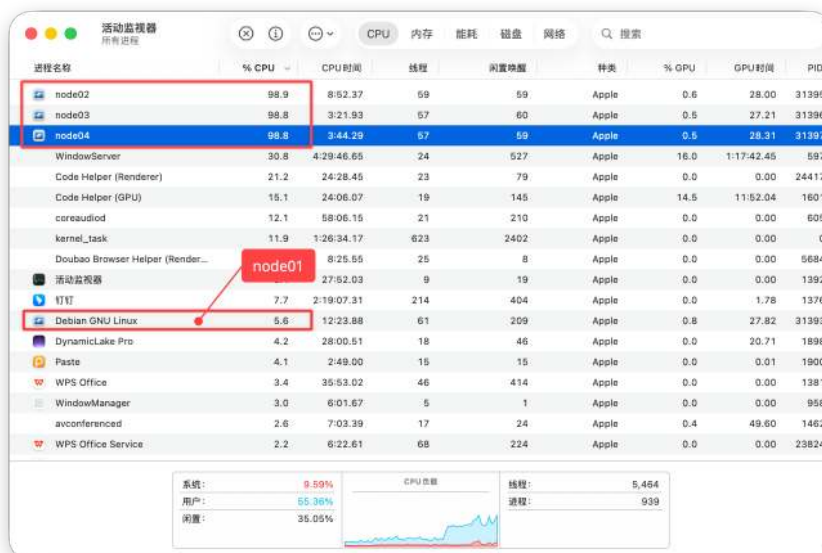
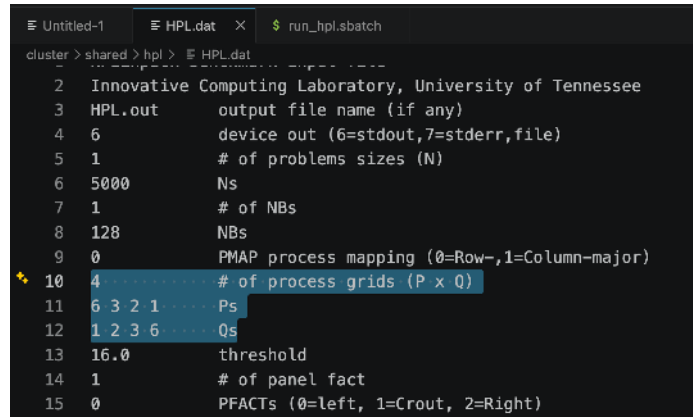


Image 25

对于调优等任务将在以后更新。

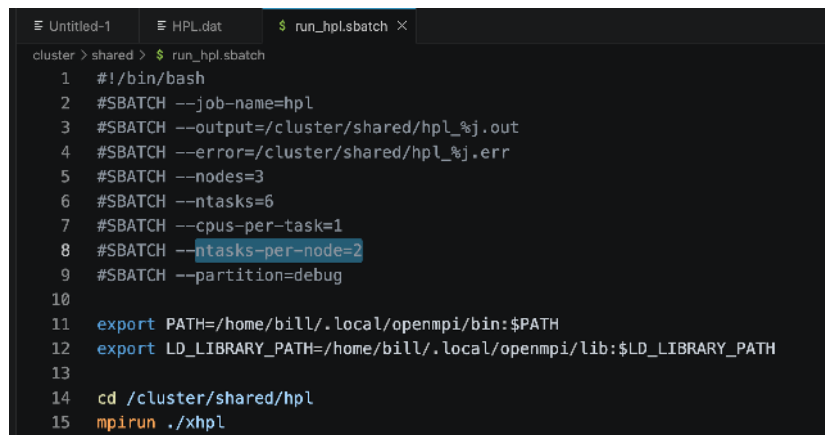
2026.7.22 更新

由于前文的 `slurm.conf` 文件里指出每个节点是2核心的 (`CPUs=2`)，所以进程网络可以调整为 `2×3` 或 `3×2`。



```
cluster > shared > hpl > HPL.dat
2 Innovative Computing Laboratory, University of Tennessee
3 HPL.out output file name (if any)
4 6 device out (6=stdout,7=stderr,file)
5 1 # of problems sizes (N)
6 5000 Ns
7 1 # of NBs
8 128 NBs
9 0 PMAP process mapping (0=Row-,1=Column-major)
10 4 3 # of process grids (P x Q)
11 6 3 2 1 Ps
12 1 2 3 6 Qs
13 16.0 threshold
14 1 # of panel fact
15 0 PFACTs (0=left, 1=Crout, 2=Right)
```

Image 26



```
cluster > shared > $ run_hpl.sbatch
1 #!/bin/bash
2 #SBATCH --job-name=hpl
3 #SBATCH --output=/cluster/shared/hpl_%j.out
4 #SBATCH --error=/cluster/shared/hpl_%j.err
5 #SBATCH --nodes=3
6 #SBATCH --ntasks=6
7 #SBATCH --cpus-per-task=1
8 #SBATCH --htasks-per-node=2
9 #SBATCH --partition=debug
10
11 export PATH=/home/bill/.local/openmpi/bin:$PATH
12 export LD_LIBRARY_PATH=/home/bill/.local/openmpi/lib:$LD_LIBRARY_PATH
13
14 cd /cluster/shared/hpl
15 mpirun ./xhpl
```

Image 27

以下是部分AI的对话。

0.0.1 那之前是两个cpu应对一个task吗，就是1*3的时候？还是说有性能浪费？

对，之前 `--ntasks=3`（每节点1进程）时，每个节点只用了 1 个核，另 1 个核空闲。对比一下：

配置	每节点进程	CPU使用率	性能
1×3 (旧)	1 进程	只用 1/2 核	~13 Gflops
2×3 (新)	2 进程	用满 2/2 核	~22-24 Gflops

Table 1

性能几乎翻倍，因为之前确实浪费了一个核。

不过现在 `--ntasks=6` + `--cpus-per-task=1` 就是完美 1:1 映射——每个进程独占一个 CPU 核心，没有浪费也没有超分，是最优配置。

0.0.2 是不是每次跑完之后要停止一段时间？不同的任务是先后关系吗？你说的“1. 顺序影响性能——第一个跑的配置往往更快”，这是为什么呢？

好问题！分析一下：

1. 不同配置之间是先后关系吗？

是的，`mpirun ./xhpl` 一次运行中，4 个 $P \times Q$ 配置是**顺序执行**的——跑完一个紧接着跑下一个，中间没有停顿。

2. 为什么第一个最快？

看你的数据趋势就很明显了：

1	测试顺序	$P \times Q$	耗时	Gflops
2	①	1×6	$28.81s \rightarrow$	23.15 🏆 最快
3	②	2×3	$30.30s \rightarrow$	22.00 ↓
4	③	3×2	$32.50s \rightarrow$	20.52 ↓
5	④	6×1	$39.64s \rightarrow$	16.82 🏆 最慢

性能**依次递减**，跟 $P \times Q$ 关系不大，关键是**越往后越慢**。

最可能的原因是 **CPU 过热降频**——这些小 ARM 设备散热有限，第一个测试时 CPU 还凉快，跑满频率；连续全速跑 2 分多钟后，温度上去就会**自动降频**，后面越来越慢。

0.1 重新跑一次，得到如下结果

为了验证是否是CPU降频的问题，我将四组数据倒序测试：

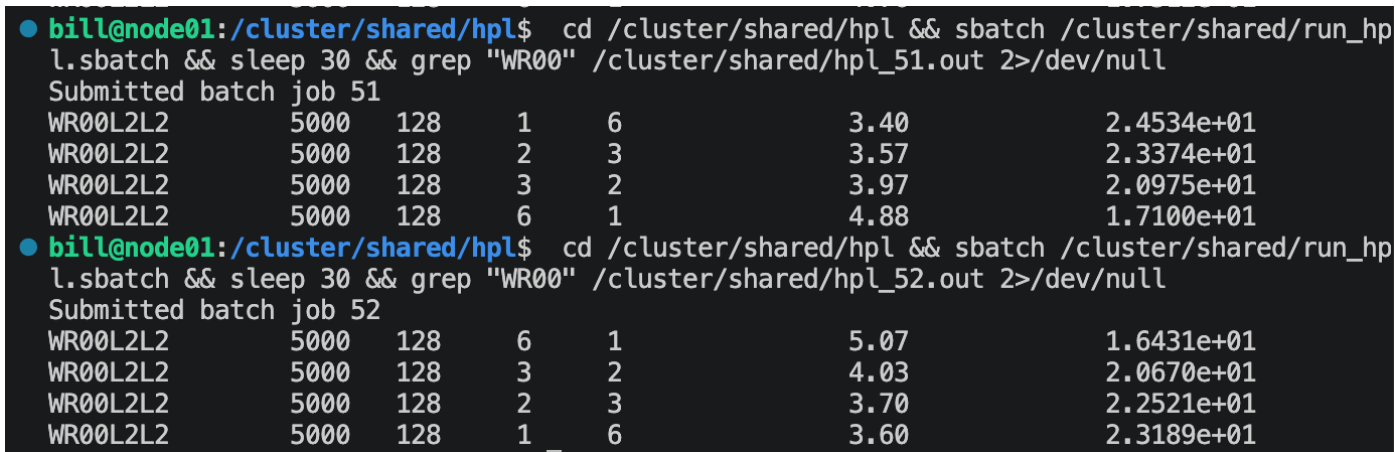


Image 28

由此可见，排名不受顺序影响—— $1 \times 6 > 2 \times 3 > 3 \times 2 > 6 \times 1$ 。

再次询问AI，“在这个小 ARM 集群上，内存访问的连续性比通信拓扑更重要。对于较大问题， 2×3 的优势才会显现出来。”

更改N的值：

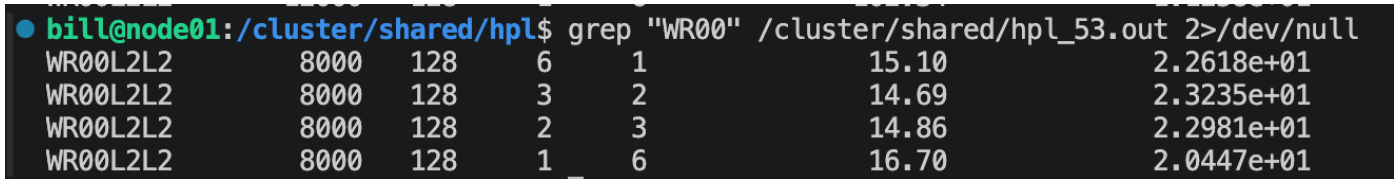


Image 29

$N=8000$ 时， 3×2 和 2×3 表现最好（约 23 Gflops），方阵网格开始显现优势， 1×6 的列连续优势在较大矩阵下已消失。

取 $P=3$, $Q=2$ ，调整分块大小 NB 以优化计算效率：

```

bill@node01:/cluster/shared/hpl$ sleep 120 && grep "WR00" /cluster/shared/hpl_55.out 2>/dev/null
WR00L2L2      8000    32    3    2      13.87      2.4623e+01
WR00L2L2      8000    64    3    2      14.08      2.4244e+01
WR00L2L2      8000    96    3    2      14.80      2.3069e+01
WR00L2L2      8000   128    3    2      15.42      2.2142e+01
WR00L2L2      8000   160    3    2      15.89      2.1489e+01
WR00L2L2      8000   192    3    2      17.10      1.9965e+01
WR00L2L2      8000   256    3    2      19.67      1.7355e+01

```

Image 30

1. Bonus: 更换数学库

这里选择了 **OpenBLAS**，因为其对于 arm 架构有比较好的支持，且安装和获取相对容易。

```

1 | cd /home/bill/OpenBLAS-0.3.28
2 | make clean
3 | make -j1 CC=gcc FC=gfortran TARGET=ARMV8 USE_OPENMP=0 NO_LAPACK=0 DYNAMIC_ARCH=0

```

接着修改 hpl 的 make 配置的库路径：

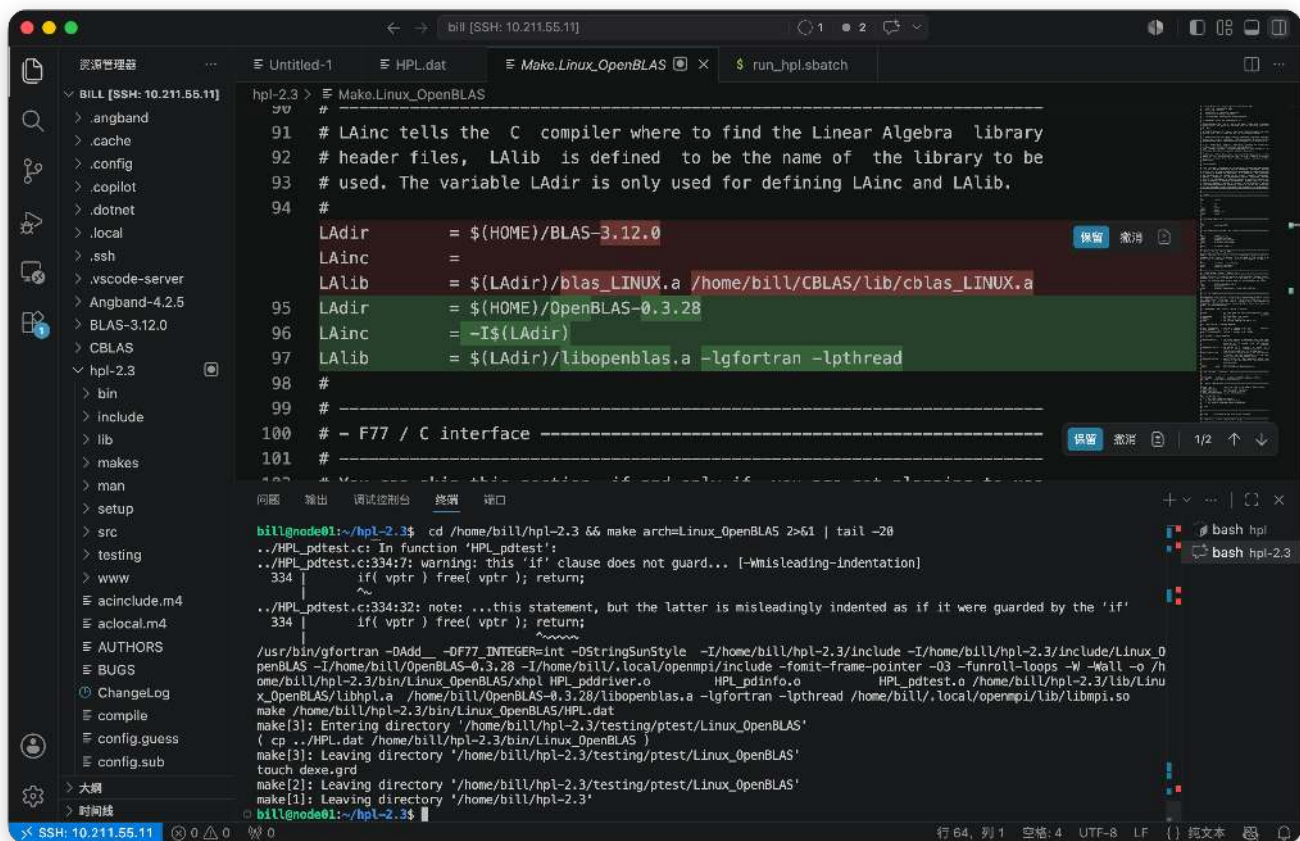


Image 31

接着创建新的 `xhpl_openblas` 文件和新的 `run_hpl_openblas.sbatch` 测试脚本。修改对应的库路径，然后开始测试，得出结果：

```
● sleep 30 && grep "WR00" /cluster/shared/hpl_57.out 2>/dev/null
WR00L2L2      8000    32    3    2      13.63      2.5059e+01

bill@node01:/cluster/shared/hpl$ cat /cluster/shared/hpl_openblas_58.out

- The matrix A is randomly generated for each test.
- The following scaled residual check will be computed:
  ||Ax-b||_oo / ( eps * ( || x ||_oo * || A ||_oo + || b ||_oo ) * N )
- The relative machine precision (eps) is taken to be 1.110223e-16
- Computational tests pass if scaled residuals are less than 16.0

=====
T/V          N    NB    P    Q          Time          Gflops
=====
WR00L2L2      8000    32    3    2          4.79          7.1237e+01
HPL_pdgesv() start time Wed Jul 22 14:01:25 2026

HPL_pdgesv() end time   Wed Jul 22 14:01:30 2026

=====
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)= 5.31096269e-03 ..... PASSED
=====
```

Image 32

- 1 ## 为什么性能提升这么大 (25 → 71 Gflops, +184%) ?
- 2
- 3 唯一的变量是数学库。参考 BLAS 是纯 Fortran 写的朴素实现，没有架构优化。OpenBLAS 包含手写的 ARM NEON 汇编内核，利用了 SIMD 向量化、缓存分块、内存预取等技术，在 ARMv8 架构上差距自然很大。

2. Bonus：更换共享文件系统

这里选择了 **JuiceFS**，因为其部署轻、无需额外服务节点、提供 aarch64 静态二进制，适合在小规模 ARM 集群上快速搭建。

```
1 # 下载
2 wget https://github.com/juicedata/juicefs/releases/download/v1.2.2/juicefs-1.2.2-
  linux-arm64.tar.gz
3 tar -xzf juicefs-1.2.2-linux-arm64.tar.gz
4 sudo cp juicefs /usr/local/bin/
5
6 # 初始化 (SQLite 元数据 + NFS 共享路径存储数据块)
7 juicefs format --storage file --bucket /cluster/shared/jfs_data
  sqlite3:///cluster/shared/jfs.db finalfs
8
9 # node01 和 node02 分别挂载
10 juicefs mount -d sqlite3:///cluster/shared/jfs.db /mnt/jfs
```

输出了 `Do not have write permission on /mnt/jfs [mount_unix.go:472]`

权限问题，node02 上 jfs 属于 root。在 node02 终端执行：

```
1 sudo chown bill:bill /mnt/jfs # 释放sudo权限
2 juicefs mount -d sqlite3:///cluster/shared/jfs.db /mnt/jfs
3 # 挂载，db是存储文件元数据的数据库后缀，/mnt/jfs是挂载目录
```

挂载成功，但是在验证跨节点通信时，却得到 `mnt/jfs/cross.txt: Input/output error` 的提示。

所以我只好又去询问万能的ai:

- 1 因为 SQLite 元数据引擎是单文件锁定模式。node01 挂载时持有 SQLite 数据库的写锁，node02 再挂载时只能以只读模式工作。当你尝试在 node02 上 `cat` 一个文件时，JuiceFS 需要从元数据库读取文件的块位置信息，但此时 SQLite 锁被 node01 持有的写操作占用，node02 无法正常访问元数据，导致 IO 错误。
- 2
- 3 简单说: **SQLite 同一时刻只允许一个客户端写入，JuiceFS 的多挂载依赖元数据引擎的并发能力。** 要真正支持多节点并发，需要用 Redis、MySQL 等支持并发的元数据引擎替代 SQLite。

```
● bill@node01:~$ cat /mnt/jfs/both_mounted.txt && echo "=== node01 writes now ===" && echo "node01 live write" > /mnt/jfs/live01.txt && ssh bill@node02 "cat /mnt/jfs/both_mounted.txt 2>&1 && echo '=== node02 writes ===' && echo 'node02 live write' > /mnt/jfs/live02.txt && cat /mnt/jfs/live02.txt" && echo "=== node01 reads node02's file ===" && cat /mnt/jfs/live02.txt
node01 wrote this - Wednesday, July 22, 2026 PM06:24:27 HKT
=== node01 writes now ===
node01 wrote this - Wednesday, July 22, 2026 PM06:24:27 HKT
=== node02 writes ===
node02 live write
=== node01 reads node02's file ===
node02 live write
● bill@node01:~$ cd /mnt/jfs/
● bill@node01:/mnt/jfs$ ls
both_mounted.txt live01.txt live02.txt
○ bill@node01:/mnt/jfs$
```

Image 33

3. Bonus: k3s

因为 k3s 需要一些额外的内存，这个时候我去更改内存大小的配置。结果 ssh 突然用不了了。具体情况是虚拟机之间能 ping 通；主机和任一虚拟机节点均无法 ping 通。我一度怀疑是我改内存配置过于着急而强制关机产生了这些问题（是的，关机还要 sudo 密码和一行命令，懒得打了直接中断运行），前前后后折腾了一个小时（我已急哭）。最后发现是 rvpn 将这部分网段的流量放进管道了。。。

```

● bill@node01:~$ ssh -t bill@node02 "wget -q0 /tmp/k3s-install.sh https://get.k3s.io && echo 'downloaded OK' && sudo sh /tmp/k3s-ins
tall.sh agent --server https://10.211.55.11:6443 --token K10cc79c5e9896cb7518499781e063247ee667bab45abfff244394697494ede6f1c::serve
r:3798150f5b8a1ca6e8aa612b32a4d538 2>&1 | tail -15"
downloaded OK
[sudo] password for bill:
[INFO] Verifying binary download
[INFO] Installing k3s to /usr/local/bin/k3s
[INFO] Skipping installation of SELinux RPM
[INFO] Creating /usr/local/bin/kubectrl symlink to k3s
[INFO] Creating /usr/local/bin/crictl symlink to k3s
[INFO] Creating /usr/local/bin/ctr symlink to k3s
[INFO] Creating killall script /usr/local/bin/k3s-killall.sh
[INFO] Creating uninstall script /usr/local/bin/k3s-agent-uninstall.sh
[INFO] env: Creating environment file /etc/systemd/system/k3s-agent.service.env
[INFO] systemd: Creating service file /etc/systemd/system/k3s-agent.service
[INFO] systemd: Enabling k3s-agent unit
Created symlink '/etc/systemd/system/multi-user.target.wants/k3s-agent.service' -> '/etc/systemd/system/k3s-agent.service'.
[INFO] Host iptables-save/iptables-restore tools not found
[INFO] Host ip6tables-save/ip6tables-restore tools not found
[INFO] systemd: Starting k3s-agent
Connection to node02 closed.
● bill@node01:~$ sleep 10 && sudo kubectl get nodes -o wide
NAME          STATUS    ROLES    AGE      VERSION    INTERNAL-IP    EXTERNAL-IP    OS-IMAGE             KERNEL-VERSION
node01        Ready    control-plane 4m6s    v1.36.2+k3s1 10.211.55.11    <none>         Debian GNU/Linux 13 (trixie) 6.12.90+deb13.1
-arm64 (arm64)  containerd://2.3.2-k3s2
node02        Ready    <none>     18s     v1.36.2+k3s1 10.211.55.12    <none>         Debian GNU/Linux 13 (trixie) 6.12.90+deb13.1
-arm64 (arm64)  containerd://2.3.2-k3s2
● bill@node01:~$ sudo kubectl create deployment nginx --image=nginx:latest --replicas=2 2>&1 && sleep 15 && sudo kubectl get pods -o
wide
deployment.apps/nginx created
NAME          READY    STATUS             RESTARTS   AGE    IP            NODE    NOMINATED NODE    READINESS GATES
nginx-6797d5487-g8d9r 0/1      ContainerCreating  0         15s    <none>        node01    <none>             <none>
nginx-6797d5487-l4ztd 0/1      ContainerCreating  0         15s    <none>        node02    <none>             <none>
● bill@node01:~$ sleep 30 && sudo kubectl get pods -o wide
NAME          READY    STATUS             RESTARTS   AGE    IP            NODE    NOMINATED NODE    READINESS GATES
nginx-6797d5487-g8d9r 1/1      Running            0          49s    10.42.0.9     node01    <none>             <none>
nginx-6797d5487-l4ztd 1/1      Running            0          49s    10.42.1.3     node02    <none>             <none>
● bill@node01:~$ sudo kubectl expose deployment nginx --port=80 --type=NodePort 2>&1 && sleep 3 && sudo kubectl get svc
service/nginx exposed
NAME          TYPE        CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
kubernetes    ClusterIP   10.43.0.1      <none>          443/TCP          5m17s
nginx         NodePort    10.43.47.241   <none>          80:30477/TCP     3s
● bill@node01:~$ wget -q0- http://10.211.55.11:30477 2>&1 | head -10
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;

```

Image 34

1 agent token 是 k3s server 生成的一个认证令牌, agent 节点用它来证明自己有权加入集群。

2
3 获取方式是在 node01 (server 节点) 上读取:

4
5 sudo cat /var/lib/rancher/k3s/server/node-token
6

1 Q: NodePort 和 ClusterIP 有什么区别? 为什么集群内部的服务能被外部访问?

2
3 以"在集群里跑一个 Web 面板让用户通过浏览器提交作业"为例:

4
5 ****ClusterIP****: 只在集群内部可访问。比如集群里的 Prometheus 监控采集 JupyterHub 的指标, 不需
要外部访问, 用 ClusterIP 就行。

6
7 ****NodePort****: 在每个节点的真实网卡上开放一个端口 (30000-32767), 外部可以通过节点 IP + 该端口
访问。比如用户要通过浏览器访问 JupyterHub, 就需要 NodePort 把服务暴露出去。

8
9 原理是 kube-proxy 在每个节点上写 iptables 规则, 做了一层 DNAT (目标地址转换):

10

```

11 | ``
12 | 用户浏览器 → http://10.211.55.11:30477 (真实节点 IP + 真实端口)
13 |         |
14 |         iptables DNAT
15 |         |
16 |         10.42.0.9:80 (Pod 虚拟 IP, 集群内部)
17 | ``
18 |
19 | Pod 的 IP 是容器网络的虚拟地址, 外部路由不到。但节点的 IP 是真实的, iptables 把外部能到达的真实地址映射到了集群内部的虚拟地址, 这就是外部能访问集群服务的原因。

```

4. 进阶挑战：身份认证 / 目录服务部署

选了 LDAP。直接在 node01 上用 apt 装 slapd。

4.1 部署过程

4.1.1 安装

```
1 | sudo apt install -y slapd ldap-utils
```

装完之后用 `dpkg-reconfigure slapd` 重新配置了一下：

- 域名： `hpclab.local`，对应 LDAP 的根节点就是 `dc=hpclab,dc=local`
- 管理员密码： `admin123`
- 组织名： HPC Lab
- 后端数据库： MDB

配完之后用 admin 密码连不上，报 `Invalid credentials (49)`。查了一下发现是 `olcRootPW` 没设对，用 `slappasswd` 生成哈希后通过 `ldapmodify` 改了一下才好。

4.1.2 创建测试用户和组

LDIF 文件（ `test-user.ldif` ）详见下方图片：

```

1 | dn: ou=people,dc=hpclab,dc=local
2 | objectClass: organizationalUnit
3 | ou: people
4 | ...
5 | member: uid=testuser,ou=people,dc=hpclab,dc=local

```

然后导入：

```
1 | ldapadd -x -D "cn=admin,dc=hpclab,dc=local" -w admin123 -f test-user.ldif
```

创建了两个 OU（people 和 groups），一个用户 testuser，一个组 hpcusers，testuser 在 hpcusers 组里。


```

● bill@node01:~$ ldapwhoami -x -D "uid=testuser,ou=people,dc=hpclab,dc=local" -w "Test@1234"
dn:uid=testuser,ou=people,dc=hpclab,dc=local
● bill@node01:~$ ldapwhoami -x -D "uid=testuser,ou=people,dc=hpclab,dc=local" -w "wrongpass"
ldap_bind: Invalid credentials (49)
● bill@node01:~$ ldapwhoami -x -D "cn=admin,dc=hpclab,dc=local" -w "admin123"
dn:cn=admin,dc=hpclab,dc=local
● bill@node01:~$ ssh node02 "python3 -c \"
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(3)
s.connect(('10.211.55.11', 389))
print('LDAP port 389: OPEN')
s.close()
\"
LDAP port 389: OPEN
● bill@node01:~$ ssh node02 "python3 -c \"
import socket
# 发送 LDAP bind 请求, 验证能实际通信
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(3)
s.connect(('10.211.55.11', 389))
# 发一个简单的 LDAP bind request 包
s.send(bytes.fromhex('300c020101600702010404008000'))
resp = s.recv(1024)
print(f'LDAP server responded: {resp.hex()}')
print('Cross-node LDAP communication: OK')
s.close()
\"
LDAP server responded: 3034020101612f0a0102040004287265717565737465642070726f746f636f6c2076657273696666
e206e6f7420737570706f72746564
Cross-node LDAP communication: OK
● bill@node01:~$

```

Image 36

4.3.1 查询

```
1 | ldapsearch -x -LLL -b "dc=hpclab,dc=local"
```

能查到所有条目, 包括 testuser 的信息和 hpcusers 组的成员关系。

4.3.2 认证

```

1 | # 正确密码
2 | ldapwhoami -x -D "uid=testuser,ou=people,dc=hpclab,dc=local" -w "Test@1234"
3 | # → dn:uid=testuser,ou=people,dc=hpclab,dc=local (成功)
4 |
5 | # 错误密码
6 | ldapwhoami -x -D "uid=testuser,ou=people,dc=hpclab,dc=local" -w "wrong"
7 | # → ldap_bind: Invalid credentials (49) (失败)

```

4.3.3 跨节点连通

从 node02 测试连 node01 的 389 端口:

```

1 | python3 -c "
2 | import socket
3 | s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
4 | s.settimeout(3)
5 | s.connect(('10.211.55.11', 389))
6 | print('LDAP port 389: OPEN')
7 | "
8 | # → LDAP port 389: OPEN

```

node02 能连上，说明 LDAP 服务对集群其他节点可用。

服务状态：

- 1
- slapd.service: active (running), enabled
- 2
- 监听: 0.0.0.0:389
- 3
- 内存: ~3MB

4.4 在集群中的作用

LDAP 在集群里做的是统一身份认证。用户登录时，SSH 的 PAM 模块会去 LDAP 查这个用户存不存在、密码对不对。这样管理员只需要在 LDAP 里建一次用户，集群所有节点都能用这个账号登录，不用每台机器都 useradd 一遍。

不过目前只是把 LDAP 服务跑起来了，还没有配置 PAM 让 node02 实际通过 LDAP 登录（需要装 libnss-ldapd 和 nslcd）。

打算继续配置：

Img37请忽略，配置过程有点混乱了

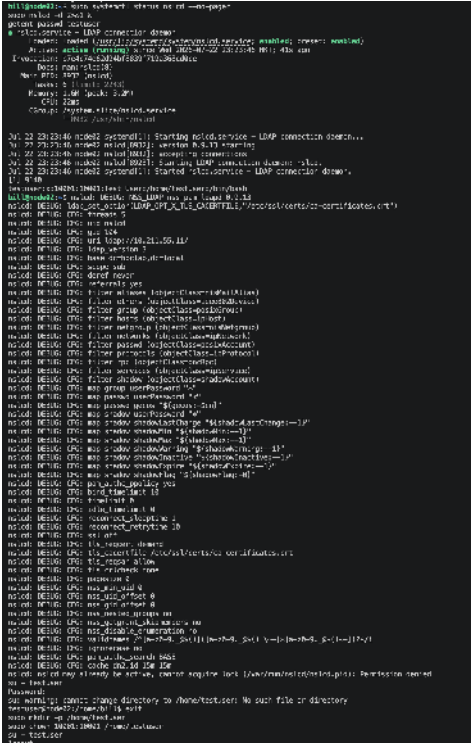


Image 37

```
● bill@node01:~$ ldapsearch -x -D "cn=admin,dc=hpclab,dc=local" -w admin123 -LLL -b "uid=testuser,ou=people,dc=hpclab,dc=local" userPassword
dn: uid=testuser,ou=people,dc=hpclab,dc=local
userPassword:: VGVzdEAxMjM0

● bill@node01:~$ echo "VGZzdEAxMjM0" | base64 -d; echo
Test@1234

○ bill@node01:~$ ssh bill@node02
Linux node02 6.12.90+deb13.1-arm64 #1 SMP Debian 6.12.90-2 (2026-05-27) aarch64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Wed Jul 22 23:32:04 2026 from 10.211.55.11
bill@node02:~$ su - testuser
Password:
testuser@node02:~$ ls
1.txt
testuser@node02:~$
```

base64解码得到密码，密码位于node01

在node02切换用户到testuser，密码输入Test@1234，成功登录

Image 38

Agent: 对，在 node02 上用 testuser 登录成功就是成功了。证明整个认证链路通了：

- 1 | node02 登录 testuser
- 2 | → PAM 查本地 /etc/passwd，没有
- 3 | → 转给 nslcd
- 4 | → nslcd 连 node01 的 LDAP (10.211.55.11:389)
- 5 | → LDAP 验证密码正确
- 6 | → 登录成功

节点	/home/testuser	说明
node01	不存在	没建过
node02	存在	刚才 sudo mkdir 建的

Table 2

LDAP 只管**账号和密码**，不管家目录。家目录是每台机器本地的，需要手动建（或者配 NFS 共享家目录，让所有节点用同一个，那就是存储服务的任务了）。